COMPUTATIONAL METHODS IN APPLIED MATHEMATICS, Vol. 8 (2008), No. 4, pp. 336–349

PERFORMANCE OF H-LU PRECONDITIONING FOR SPARSE MATRICES

L. GRASEDYCK[1] W. HACKBUSCH[2], AND R. KRIEMANN[2]

Abstract — In this paper we review the technique of hierarchical matrices and put it into the context of black-box solvers for large linear systems. Numerical examples for several classes of problems from medium- to large-scale illustrate the applicability and efficiency of this technique. We compare the results with those of several direct solvers (which typically scale quadratically in the matrix size) as well as an iterative solver (algebraic multigrid) which scales linearly (if it converges in $\mathcal{O}(1)$ steps).

2000 Mathematics Subject Classification: 65F05, 65F30, 65F50, 65N55.
Keywords: hierarchical matrices, black-box clustering, preconditioning, LU, direct solver, iterative solver, performance.

1. Introduction

Many methods for the fast solution of large-scale linear systems exploit some hierarchical structure of the system. For example, when the system stems from the finite element (FE) discretization of a partial differential equation (PDE) posed in a domain $\Omega \subset \mathbb{R}^d$, then one can exploit the knowledge of the underlying geometrical problem. Whereas multigrid methods use a discretization of the pde on different levels of resolution, hierarchical ($\mathcal{H}$-) matrices use the geometry information associated to the degrees of freedom on a single level in order to find a distance measure for degrees of freedom. Once this distance measure is available, an almost black-box type arithmetic can be applied for the inversion or factorization of the system matrix.

The structure and arithmetic of $\mathcal{H}$-matrices was first introduced in 1998 [20] for a one-dimensional problem. In this structure it is possible to assemble, store and evaluate dense matrices in almost linear complexity in the size of the matrix, i.e., with a complexity of $\mathcal{O}(n \log(n)^c)$ for $n \times n$ matrices and a small constant c. This extends to higher dimensions $d = 2, 3$ with a moderate constant c [15, 21].

The key idea for the approximation of dense matrices, e.g. the triangular factors of an LU-decomposition of the system matrix, is to reorder the matrix rows and columns so that certain subblocks of the reordered matrix can be approximated by low-rank matrices. These low-rank matrices can be represented by a product of two rectangular matrices as follows: Let $A \in \mathbb{R}^{m \times m}$ with rank(A)=k and $k \ll m$. Then there exist matrices $B, C \in \mathbb{R}^{m \times k}$ such that $A = BC^\top$. Whereas A has m^2 entries, B and C, when taken together, have $2km$ entries which results in significant savings in storage if $k \ll m$.

[1] *Berlin Mathematical School, TU Berlin*, Straße des 17. Juni 136, 10623 Berlin, Germany. E-mail: lgr@mis.mpg.de

[2] *Max Planck Institute for Mathematics in the Sciences*, Inselstraße 22–26, 04103 Leipzig, Germany. E-mail: {wh,rok}@mis.mpg.de

In finite element methods, the stiffness matrix is sparse but its LU factors are fully populated and can be approximated by an $\mathcal{H}$-matrix. Such approximate $\mathcal{H}$-LU factors may then be used as a preconditioner in iterative methods [8, 17, 18, 26] or as a direct solver.

In most of the previous papers on $\mathcal{H}$-matrices [7, 8, 15, 18], the construction of the $\mathcal{H}$-matrix block structure is based on geometry information associated with the underlying indices. Each index is associated with its basis function and a (rectangular) bounding box of the support of the basis function. The standard geometric clustering algorithms, which include the bisection as well as the nested dissection clustering, compute Euclidean diameters and distances based on these geometric entities in order to construct the block partition of the $\mathcal{H}$-matrix format. More recently, an algebraic clustering algorithm has been introduced which is applicable to *sparse* matrices and only needs the matrix itself as input [8, 16, 17, 25, 27]. A matrix graph is constructed based on the sparsity structure of the matrix, and the subsequent algebraic clustering algorithm is based on this matrix graph. One obtains an algebraic algorithm for constructing an $\mathcal{H}$-matrix construction that is similar to algebraic multigrid (AMG) techniques [9, 10, 19, 23, 29].

Given an $\mathcal{H}$-matrix format, we can convert a sparse matrix into an $\mathcal{H}$-matrix and compute its $\mathcal{H}$-LU factorization. This yields a preconditioner to accelerate the iterative solution of the linear system of equations. We will apply this preconditioner in the iterative solution of several benchmark linear systems, providing comparisons with the direct solvers (Pardiso [30–32], Umfpack [11], SuperLU [12], Spooles [4], Mumps [1–3]) and an iterative solver (Boomer-AMG [23]).

The remainder of this paper is structured as follows: In Section 2 we provide a brief introduction to $\mathcal{H}$-matrices. Section 3 reviews the algebraic clustering algorithm, and in Section 4 we introduce the standard direct and iterative solvers used. In Section 5 we present numerical results for the $\mathcal{H}$-matrix approach in comparison with standard direct solvers and an iterative solver for a set of test problems.

2. Preliminaries: The model problem and $\mathcal{H}$-matrices

2.1. The model problem. Throughout this paper, we consider a linear system of equations of the form $Ax = b$, where $A \in \mathbb{R}^{N\times N}$ is sparse and invertible. To each index i from the index set $\mathcal{I} := \{1, \dots, N\}$ we assign geometric entities which are required in the original $\mathcal{H}$-matrix constructions but will no longer be required for the new black-box clustering approach (Section 3).

Definition 2.1 (Geometric entities, cluster). Let $d \in \mathbb{N}$. For each index $i \in \mathcal{I}$ we assign a (fixed) set and nodal point $\xi_i \in \Xi_i \subset \mathbb{R}^d$. The subset $v \subset \mathcal{I}$ of the index set is called a *cluster*. Its support is defined by

$$\Xi_v := \bigcup_{j\in v} \Xi_j. \tag{2.1}$$

The geometric $\mathcal{H}$-matrix construction (see subsection 2) needs (upper bounds of) the diameters of these clusters as well as the distances between two such clusters (both in the Euclidean norm). Since the diameters and distances can be computed much more efficiently for rectangular boxes than for arbitrarily shaped domains, we supply each cluster v with a bounding box

$$B_v = \bigotimes_{j=1}^{d} [\alpha_{v,j}, \beta_{v,j}] \tag{2.2}$$

that contains Ξ_v, i.e., $\Xi_v \subset B_v$.

In a finite element context, the subset $\Xi_i \subset \mathbb{R}^d$ is the support of the i-th basis function.

2.2. A brief introduction to $\mathcal{H}$-Matrices. In this section, we will review $\mathcal{H}$-matrices and their arithmetic. An $\mathcal{H}$-matrix provides a data-sparse approximation to a dense matrix by replacing certain blocks of the matrix by matrices of low rank which can be stored very efficiently. The blocks which allow for such low rank representations are selected from the hierarchy of partitions organized in the so-called cluster tree.

Definition 2.2 (Cluster tree). Let $T_\mathcal{I} = (V, E)$ be a tree with a vertex set V and an edge set E. For a vertex $v \in V$, we define the set of successors (or sons) of v as $S(v) := \{w \in V \mid (v, w) \in E\}$. The tree $T_\mathcal{I}$ is called the cluster tree of $\mathcal{I}$ if its vertices consist of subsets of $\mathcal{I}$ and satisfy the following conditions (cf. Figure 2.1 (left)):

1) $\mathcal{I} \in V$ is the root of $T_\mathcal{I}$, and $v \subset \mathcal{I}$, $v \neq \varnothing$, for all $v \in V$;

2) For all $v \in V$, there holds $S(v) = \varnothing$ or $v = \dot{\bigcup}_{w \in S(v)} w$.

The depth of the cluster tree, $d(T_\mathcal{I})$, is defined as the length of the longest path in $T_\mathcal{I}$. In the following, we identify V and $T_\mathcal{I}$, i.e., we write $v \in T_\mathcal{I}$ instead of $v \in V$. The nodes $v \in V$ are called clusters. The nodes with no successors are called *leaves* and define the set $\mathcal{L}(T_\mathcal{I}) := \{v \in T_\mathcal{I} \mid S(v) = \varnothing\}$.

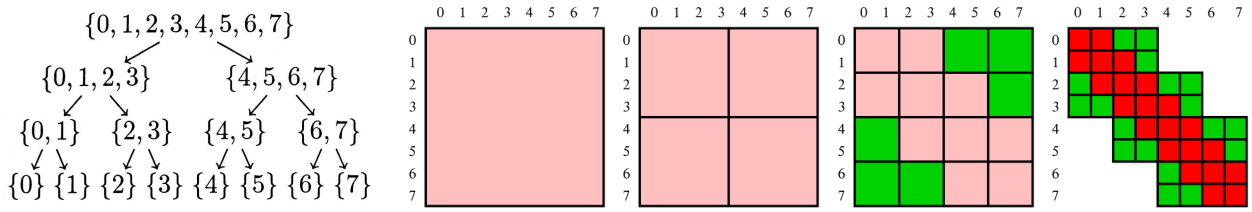

Fig. 2.1. Left: Cluster tree $T_\mathcal{I}$. Right: The four levels of the block cluster tree $T_{\mathcal{I}\times\mathcal{I}}$, where nodes that are further refined are light grey, inadmissible leaves are dark grey, and admissible leaves are grey

In previous papers, several strategies have been introduced to construct a cluster tree from a given index set, e.g., bisection or nested dissection, but most of these constructions are based on the underlying geometric entities defined in Definition 2.1. As an example, we will review the geometric bisection clustering. Here, a cluster v with support Ξ_v (2.1) is subdivided into two smaller clusters v_1, v_2 as follows.

1. Let Q_v denote a box that contains all nodal points $(\xi_i)_{i\in v}$ (cf. Definition 2.1). For the root cluster this could be the bounding box $Q_\mathcal{I} := B_\mathcal{I}$.

2. Subdivide the box Q_v into two boxes $Q_v = Q_1 \dot{\cup} Q_2$ of equal size.

3. Define the two successors $S(v) = \{v_1, v_2\}$ of v by

$$v_1 := \{i \in v \mid \xi_i \in Q_1\}, \quad v_2 := \{i \in v \mid \xi_i \in Q_2\}$$

and use the boxes $Q_{v_1} := Q_1$, $Q_{v_2} := Q_2$ for the further subdivision of the sons.

The subdivision is typically performed so that the resulting diameters of the boxes associated with successor clusters become as small as possible. A single step of geometric bisection is illustrated in Fig. 2.2 where a cluster v consisting of 17 vertices is subdivided into clusters v_1, v_2 consisting of 8 and 9 vertices lying in Q_{v_1} and Q_{v_2}, respectively. Here, the subdivision into v_1 and v_2 is based on the geometric locations associated with the indices.

Given a cluster tree $T_\mathcal{I}$, any two clusters $s, t \in T_\mathcal{I}$ form a product $s \times t$, also called a *block cluster*, which can be associated with the corresponding matrix block $(A_{ij})_{i\in s, j\in t}$ (cf. Figure 2.1 (right)). We will use an admissibility condition to decide whether such a block will be

allowed in a block partition of the matrix A or will be further subdivided. In general, an admissibility condition is a Boolean function $\mathrm{Adm} : T_{\mathcal{I}} \times T_{\mathcal{I}} \to \{\mathtt{true}, \mathtt{false}\}$.

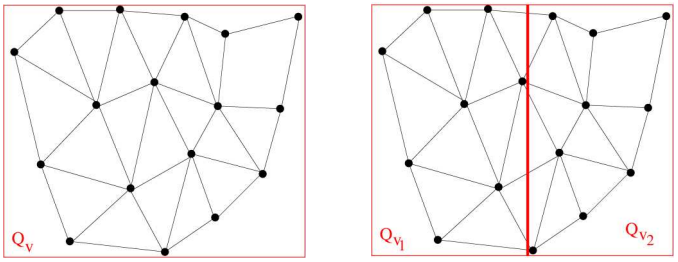

Fig. 2.2. Geometric bisection

For cluster trees based on the underlying geometry, typical admissibility conditions use geometric information, e.g., the standard admissibility condition is given by

$$\mathrm{Adm}_{\mathrm{S}}(s,t) = \mathtt{true} \quad :\Leftrightarrow \quad \min(\mathrm{diam}(B_s), \mathrm{diam}(B_t)) \leqslant \eta \ \mathrm{dist}(B_s, B_t) \tag{2.3}$$

for some $0 < \eta$. Here, B_s, B_t are the bounding boxes (2.2) of the clusters s, t, respectively, and the distance and diameters are computed with respect to the Euclidean norm.

Given a cluster tree $T_{\mathcal{I}}$ and an admissibility condition, we construct a hierarchy of block partitionings of the product index set $\mathcal{I} \times \mathcal{I}$. The hierarchy forms a tree structure and is organized in a *block cluster tree* $T_{\mathcal{I}\times\mathcal{I}}$:

Definition 2.3 (Block cluster tree). *Let $T_{\mathcal{I}}$ be a cluster tree of the index set $\mathcal{I}$. A cluster tree $T_{\mathcal{I}\times\mathcal{I}}$ is called a block cluster tree (based upon $T_{\mathcal{I}}$) if for all $v \in T_{\mathcal{I}\times\mathcal{I}}$ there exist $s, t \in T_{\mathcal{I}}$ such that $v = s \times t$. The nodes $v \in T_{\mathcal{I}\times\mathcal{I}}$ are called block clusters.*

A block cluster tree may be constructed from a given cluster tree in the canonical way defined by Algorithm 2.1 (cf. Fig. 2.1) which we will employ for all cluster trees constructed in this paper.

Algorithm 2.1 Canonical block cluster tree construction

```
procedure bct_construct(s, t, Adm(·,·), n_min)
    if Adm(s,t) = true ∨ min{#s, #t} ⩽ n_min then S(s × t) := ∅;
    else
        for all s' ∈ S(s) do
            for all t' ∈ S(t) do
                S(s × t) := S(s × t) ∪ {bct_construct(s', t', Adm(·,·), n_min)};
            end for
        end for
    end if
    return s × t;
end
```

For rather small blocks, the matrix arithmetic of a full matrix is more efficient than that of a structured matrix. Therefore, we introduce a parameter $n_{\min} \geqslant 10$: matrices are not further subdivided if they have less than or equal to $n_{\min}$ rows or columns.

The leaves of the block cluster tree obtained through this construction yield a disjoint partition of the product index set $\mathcal{I} \times \mathcal{I}$.

In Fig. 2.1, we provide a simple example for the cluster tree and the corresponding block cluster tree. The indices in this example correspond to the continuous, piecewise linear basis functions of a regularly refined unit interval.

Matrix blocks which correspond to admissible block clusters will be approximated in a data-sparse format by the following Rk-matrix representation.

Definition 2.4 (Rk-matrix representation). Let k, n, $m \in \mathbb{N}_0$. Let $M \in \mathbb{R}^{n\times m}$ be a matrix of at most rank k. A representation of M in factorized form

$$M = AB^\top, \quad A \in \mathbb{R}^{n\times k}, \quad B \in \mathbb{R}^{m\times k}, \tag{2.4}$$

with A and B stored in full matrix representation, is called an Rk-matrix representation of M, or, in short, we call M an Rk-matrix.

If the rank k is small compared to the matrix size given by n and m, we obtain considerable savings in the storage and work complexities of an Rk-matrix compared to a full matrix [15].

Finally, we can introduce the definition of a hierarchical matrix.

Definition 2.5 ($\mathcal{H}$-matrix). Let $k, n_{\min} \in \mathbb{N}_0$. The set of $\mathcal{H}$-matrices induced by a block cluster tree $T := T_{\mathcal{I}\times\mathcal{I}}$ with blockwise rank k and minimum block size $n_{\min}$ is defined by

$$\mathcal{H}(T,k) := \{M \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}} \mid \forall s\times t \in \mathcal{L}(T) : \operatorname{rank}(M|_{s\times t}) \leqslant k \text{ or } \min\{\#s, \#t\} \leqslant n_{\min}\}. \tag{2.5}$$

Blocks $M|_{s\times t}$ with $\operatorname{rank}(M|_{s\times t}) \leqslant k$ are stored as Rk-matrices whereas all other blocks are stored as full matrices.

Whereas the classical $\mathcal{H}$-matrix uses a fixed rank for the Rk-blocks, it is possible to replace it by *variable (or adaptive) ranks* in order to enforce a desired relative accuracy ε within individual blocks [15].

2.3. Arithmetic of $\mathcal{H}$-matrices. Given two $\mathcal{H}$-matrices $A, B \in \mathcal{H}(T,k)$ based on the same block cluster tree T, i.e., with the same block structure, the exact sum or product of these two matrices will typically not belong to $\mathcal{H}(T,k)$. In the case of matrix addition, we have $A + B \in \mathcal{H}(T,2k)$; the rank of an exact matrix product is less obvious. We will use a truncation operator $\mathcal{T}^{\mathcal{H}}_{k\leftarrow k'}$ to define the $\mathcal{H}$-matrix addition $C := A \oplus_{\mathcal{H}} B$ and $\mathcal{H}$-matrix multiplication $C := A \otimes_{\mathcal{H}} B$ such that $C \in \mathcal{H}(T,k)$.

The truncation of a rank k' matrix R to rank $k < k'$ is defined as the best approximation with respect to the Frobenius (or spectral) norm in the set of rank k matrices. In the context of $\mathcal{H}$-matrices, we use such truncations for all admissible (rank k') blocks. Using truncated versions of the QR-decomposition and singular value decomposition, the truncation of a rank k' matrix $R \in \mathbb{R}^{n,m}$ (given in the form $R = AB^\top$ where $A \in \mathbb{R}^{n,k'}$ and $B \in \mathbb{R}^{m,k'}$) to a lower rank can be computed with complexity $\mathcal{O}\left((k')^2(n+m)\right)$; further details are provided in [15]. We then define the $\mathcal{H}$-matrix addition and multiplication as follows:

$$A \oplus_{\mathcal{H}} B := \mathcal{T}^{\mathcal{H}}_{k\leftarrow 2k}(A+B); \qquad A \otimes_{\mathcal{H}} B := \mathcal{T}^{\mathcal{H}}_{k\leftarrow k'}(A\cdot B)$$

where $k' \leqslant c(p+1)k$ is the rank of the exact matrix product, c denotes some constant (which depends on the block cluster tree T) and p denotes the depth (Definition 2.2) of the tree. Estimates that show that the $\mathcal{H}$-matrix addition and multiplication have almost optimal complexity for typical $\mathcal{H}$-structures are provided in [15] along with details on the efficient implementation of these operations. The $\mathcal{H}$-matrix addition and multiplication are operations required to define an $\mathcal{H}$-inversion as well as an $\mathcal{H}$-LU factorization recursively in the block structure. Details on these algorithms can be found in [5, 6, 15, 18].

3. Black-box clustering for sparse matrices

So far, we have introduced $\mathcal{H}$-matrices based on the partitioning of the underlying geometry. If such geometrical data are not available, as is the case in many applications, the previous definition of the block cluster tree and the corresponding admissibility condition cannot be used for constructing $\mathcal{H}$-matrices.

In [8, 16], it was demonstrated, how to build a cluster tree by using only the sparse matrix and, therefore, how to provide purely algebraic $\mathcal{H}$-matrix arithmetics. The basic idea is to partition the (symmetrized) matrix graph $\mathcal{G}(A) = (V_A, E_A)$ with $V_A = \mathcal{I}$ and $E_A := \{(i,j) \in \mathcal{I} \times \mathcal{I} \mid i \neq j \wedge A_{ij} \neq 0\}$ by standard graph partitioning algorithms. This is justified by the observation that geometrical properties, e.g., the distance between grid points, translate into algebraic properties in the matrix graph, e.g., the length of the path between nodes, which correspond to the indices.

The proposed graph partitioning technique in [16] is a modification of the algorithm described in [14] and is based on the *breadth first search* (BFS). First, a random node v in $\mathcal{G}(A)$ is chosen and a node with a maximal distance in terms of the path length is determined using BFS. Afterwards, this process is repeated with the computed node, until the path length stagnates, yielding two nodes in $\mathcal{G}(A)$ with a (almost) maximal distance. Then, a simultaneous BFS is started at these nodes, assigning layer by layer of unvisited nodes to the corresponding two subdomains, which finally form a partition of $\mathcal{G}(A)$. For constructing the cluster tree, the process is repeated for the restricted graph defined by each subdomain.

If $\mathcal{G}(A)$ is not connected, a BFS would not succeed in visiting each node in $\mathcal{G}(A)$. Fortunately, in such a case, individual components of $\mathcal{G}(A)$ can be computed and directly used for graph partitioning, e.g., each component forms a subdomain of the partitioning. Since no connection is present between the components in the graph, the matrix can then be ordered in block diagonal form.

For the definition of an admissibilty condition neccessary for the construction of the block cluster tree, two versions are proposed in [16]. The *weak black-box* admissibility is defined by direct reachability over exactly one edge in $\mathcal{G}(A)$. In contrast to this, the *standard black-box* admissibility uses path lengths to define the diameter and distance of a cluster or between clusters, respectively (see [16, Definition 8] for details).

Beside this BFS-based method, other graph partitioning algorithms implemented in corresponding software libraries, e.g., METIS [24] and Scotch [28], were examined in [16] for the applicability to $\mathcal{H}$-matrices. Especially the multilevel method from METIS was found to yield robust and efficient $\mathcal{H}$-arithmetic in terms of $\mathcal{H}$-LU preconditioning.

The idea of algebraic clustering can be further enhanced in the context of nested dissection (see [17] and [16]), providing more efficiency in the $\mathcal{H}$-LU factorization for sparse matrices and therefore faster preconditioning. It should be noted that there is a crucial difference between the multilevel nested dissection used in direct solvers and the corresponding ordering for $\mathcal{H}$-matrices: the vertex separator between the subdomains in nested dissection has to be further partitioned by graph bisection techniques to form a corresponding cluster tree (see [16, Sec. 3.3]). Otherwise, a large dense block has to be used in the $\mathcal{H}$-matrix, destroying the almost linear memory and time complexity.

Due to the results in [16], in this comparison METIS was used for graph partitioning in the black-box $\mathcal{H}$-matrix technique. The partitioning of the vertex separator is performed by a special version of the BFS-based algorithm described in [16].

4. Direct and iterative solvers for sparse matrices

To assess the performance of $\mathcal{H}$-matrices for solving large sparse linear systems, we compared them to widely used and well respected standard solver packages. Here, the majority falls into the class of direct solvers for symmetric and non-symmetric systems. Such solvers are used in many applications where medium-scale linear systems have to be solved without a priori knowledge about the system. Often, these direct solvers are implemented very efficiently and provide optimal flop rates on high performance machines.

Although $\mathcal{H}$-matrix based solvers can also be used as direct solvers, it is usually much more efficient to use them in an iterative scheme, e.g., GMRES preconditioned by an $\mathcal{H}$-LU factorization. This is also the case for algebraic multigrid methods, of which one representative is chosen to be part of this comparison.

Most of the below described solver packages can be optimised for a particular sparse system to be solved, e.g., by choosing different index permutations or thresholds. In this comparison all solvers are seen as a black box, i.e., no knowledge of the linear system is given by the user except the system itself. All parameters to the solver packages are set to the default values suggested by the package maintainers.

4.1. Direct solvers. The direct solvers described in this section are all based on computing a pivot sequence for an efficient factorization (being either Cholesky, LDL^T or LU) of a sparse matrix. For computing the ordering in the first stage of the factorization process, different methods are employed. The most common algorithms are the *approximate minimum degree* or *AMD* ordering [1] and the (multilevel) *nested dissection* ordering [24] as is also applied in the algebraic $\mathcal{H}$-matrix technique.

This symbolic factorization is followed by the actual numerical factorization process. There, again different strategies are possible, e.g., left-looking, right-looking or multifrontal algorithms. For a detailed introduction into sparse matrix solvers, we refer to [22] and the references therein. Similarly to $\mathcal{H}$-matrices, the factorization phase is the most time consuming step in solving the system. In contrast to this, the time for the solve phase can often be neglected, especially for large systems. When solving for many right-hand sides, however, it plays a vital role.

For our comparison we use the following direct solvers:

UMFPACK (see [11]) by Davis et al. is a sparse direct solver written in C for non-symmetric systems. It is freely available with an open source license. An earlier version is used in Matlab and Mathematica to solve sparse systems, indicating the quality and efficiency of this solver. By default, UMFPACK uses an AMD strategy for ordering the indices and a multifrontal approach for factorization.

PARDISO (see [30] and [31]) developed by Schenk and Gärtner uses the multilevel nested dissection technique based on orderings computed by the METIS graph partitioning library [24]. Alternatively, a fill-reducing ordering can be employed. The actual factorization is performed with a combination of left- and right-looking techniques. The solver library is available for academic and commercial use in binary form.

MUMPS (see [1–3]) from Amestoy et al. is designed to solve symmetric and non-symmetric sparse linear systems. It supports a wide range of different ordering algorithms, e.g., AMD, approximate minimum fill-in and (multilevel) nested dissection. The latter is performed by using the METIS graph partitioning library. By default, MUMPS will

automatically choose an ordering technique depending on the available packages, e.g., with or without METIS. MUMPS is freely available under a public domain license with full source code.

SuperLU (see [12]) by Demmel et al. uses a partial (threshold) pivot search combined with a supernodal approach for non-symmetric systems to compute the LU factorization of a sparse matrix. All ordering techniques are implemented in SuperLU without the need for external software packages. The source code for SuperLU is freely available.

Spooles (see [4]) by Ashcraft et al. uses multiple minimum degree, nested dissection and multisection orderings to compute factorizations for symmetric and non-symmetric sparse matrices. The factorization itself is performed with a left-looking algorithm. Spooles is freely available in source form.

4.2. Iterative solvers. The iterative solvers that we use in our comparison are a GMRES-accelerated algebraic multigrid method and an $\mathcal{H}$-LU preconditioned GMRES. Both of these use as an outer iteration the GMRES algorithm and perform different preconditioning techniques.

4.2.1. BoomerAMG. The AMG solver BoomerAMG is part of the more general solver package *Hypre*, `https://computation.llnl.gov/casc/hypre/software.html`. Geometric multigrid methods are highly efficient solvers for partial differential equations, assuming that the convergence properties are fulfilled. The general idea of the multigrid technique is to have a hierarchy of grids and corresponding linear systems and to obtain corrections for the current approximation on a finer grid by solving the system on a coarser grid. Usually, one or more so-called smoothing steps are necessary to be able to effectively project the vectors between grid levels. Such projection operators (prolongation and restriction) are highly dependent on the grid hierarchy, the linear system, and the underlying discretization space.

Algebraic multigrid methods try to perform the same operations without geometric information. Coarse grid matrices and the corresponding projections are constructed purely based on the supplied sparse matrix.

Multigrid methods can be used as an iterative solver. Alternatively, they are employed as preconditioners in a conjugate gradient (CG) or generalised minimal residual (GMRES, with restart) iteration. The latter is also used in this comparison, i.e., the convergence of AMG is accelerated or enforced by GMRES. Furthermore, ILU smoothing in the form of the supplied Euclid algorithm (see `https://computation.llnl.gov/casc/hypre/software.html`) is chosen during the multigrid process. Although this choice increases the setup time, it is more robust than a simple Gauss — Seidel or Jacobi smoother.

It is not expected that algebraic multigrid methods in general or BoomerAMG in particular are capable of handling all test systems in this comparison. We have chosen an AMG solver to show the difference between the $\mathcal{H}$-matrix technique and algebraic multigrid methods in the cases where the latter performs reasonably well.

4.2.2. $\mathcal{H}$-LU preconditioning. Similarly to the algebraic multigrid technique, $\mathcal{H}$-matrices are used as preconditioners in a GMRES iteration to solve the sparse linear system. In the optimal case the accuracy of the $\mathcal{H}$-LU factorization is set such that $\|I-(LU)^{-1}A\| \leqslant \rho < 1$, where L and U are the $\mathcal{H}$-LU factors of the given sparse matrix A, therefore ensuring convergence of the iterative process with a rate of at least ρ. In practice however, this is not

always possible to achieve with reasonable costs, and fortunately not necessary for GMRES to converge. Therefore, by default an accuracy of $\varepsilon = 10^{-4}$ is used for the $\mathcal{H}$-arithmetic.

5. Numerical results

We are interested in two main aspects in this study: applicability of the $\mathcal{H}$-matrix technique for a variety of problems and the scaling behavior of different solvers with respect to the problem size. For the latter, different partial differential equations are to be solved with different numbers of unknows. In particular, the Poisson equation

$$-\Delta u = f, \tag{5.1}$$

the Helmholtz equation

$$-\Delta u + \lambda u = f, \tag{5.2}$$

and the convection-diffusion equation

$$-\kappa \Delta u + b \cdot \nabla u = f \tag{5.3}$$

with $\kappa = 10^{-3}$ and therefore dominant convection $b(x) := (0.5 - x_2, x_1 - 0.5)^\top$ are used as benchmark problems. Each is computed in $\Omega = [0,1]^d$, $d = 2, 3$. In $\mathbb{R}^3$, the third component of b is zero.

For the applicability test, several matrices from the *University of Florida Sparse Matrix Collection* (http://www.cise.ufl.edu/research/sparse/matrices) maintained by Tim Davis were chosen. The test matrices cover different matrix types (e.g., unsymmetric, positive definite, indefinte) from different applications (e.g., semiconductor device simulation, quantum chemistry, financial mathematics, eigenvalue problems) and different problem sizes, where the interest is especially in larger systems ($\gtrsim 50,000$ unknowns).

All numerical tests were performed on a Sun X4600 with 8 dual core Opteron processors running at 2.8 GHz and 196 GB of main memory. All solvers with access to the source code were compiled with full optimization.

The computing time in each experiment is measured in seconds. Memory usage is in MB and read directly from the operating system as the amount of memory the corresponding process occupies without static memory of the program. Furthermore, the initial matrix in compressed row or compressed column format is also not taken into account. Therefore, only memory allocated directly by the solver or needed due to special data formats is measured.

It should be noted that a Jacobi preconditioned GMRES iteration simply did not converge (in reasonable time) for most of the test problems; it is therefore excluded for this comparison.

5.1. Scaling behavior. First, the complexity properties with respect to the factorization and solving time and memory consumption of each solver is examined for the test problems (5.1), (5.2), and (5.3), respectively. The allowable solution time for each solver is restricted to $60,000$ seconds.

For the Poisson equation (5.1) in $d = 2$ the results are shown in Fig. 5.1. It is obvious, that the preprocessing time for the algebraic multigrid method is much shorter than for all other methods. Furthermore, the linear complexity is clearly visible compared to the linear-logarithmic complexity for the direct solvers or the $\mathcal{H}$-matrix solver. Nevertheless, especially PARDISO and MUMPS show an almost optimal scaling behavior, only slightly trailed by UMFPACK. $\mathcal{H}$-matrices, although theoretically better in terms of complexity,

only seem to be gaining little ground compared to the fastest direct solvers. Obviously, the problem sizes are not large enough to compensate the larger constants involved in the $\mathcal{H}$-matrix arithmetics. Only SuperLU and Spooles show a higher complexity.

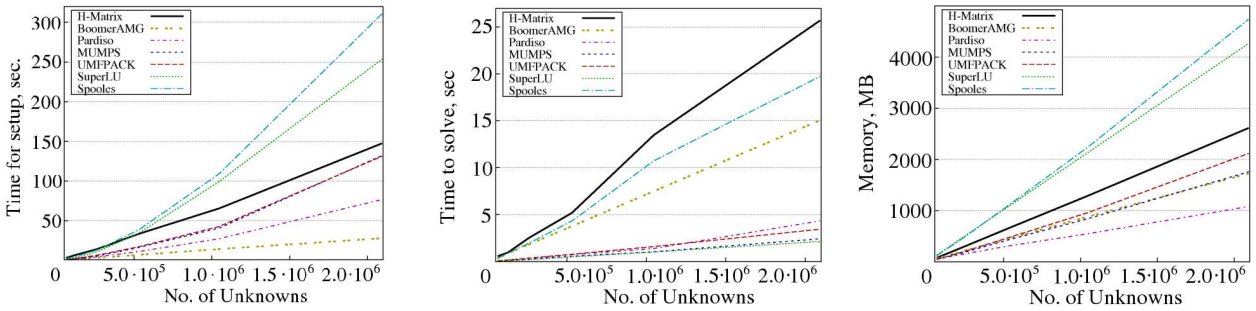

Fig. 5.1. Complexity of setup (left), solve (middle) and memory (right) for the Poisson equation in $\mathbb{R}^2$

A similar picture is shown for the solving phase. Since only a single forward elimination plus backward substitution is needed for direct solvers, they clearly show the shortest execution time. BoomerAMG has a disadvantage in this phase, because an iterative process is involved. Hence, for a large number of right-hand sides, the preprocessing advantage is lost compared to direct solvers. The same holds for the $\mathcal{H}$-matrix solver. Remarkable are the high costs for the Spooles solver, indicating efficiency problems.

In terms of memory consumption, the same behavior can be seen as in the factorization phase. UMFPACK, MUMPS, PARDISO and BoomerAMG show a very similar memory consumption, followed by $\mathcal{H}$-matrices and SuperLU and Spooles.

For the Helmholtz equation (5.2) in $d = 2$ the situation is a little bit different. First, the algebraic multigrid method did not converge for this test example within 1000 iteration steps, hence, results are not included. Second, the better complexity of the direct solvers is higher, whereas the $\mathcal{H}$-matrix solver needed about the same time compared to the Poisson problem. Hence, the point of break even is reached earlier. In fact, only UMFPACK is able to outperform $\mathcal{H}$-matrices for all problem sizes. Again, SuperLU and Spooles show a significantly higher complexity (see Fig. 5.2).

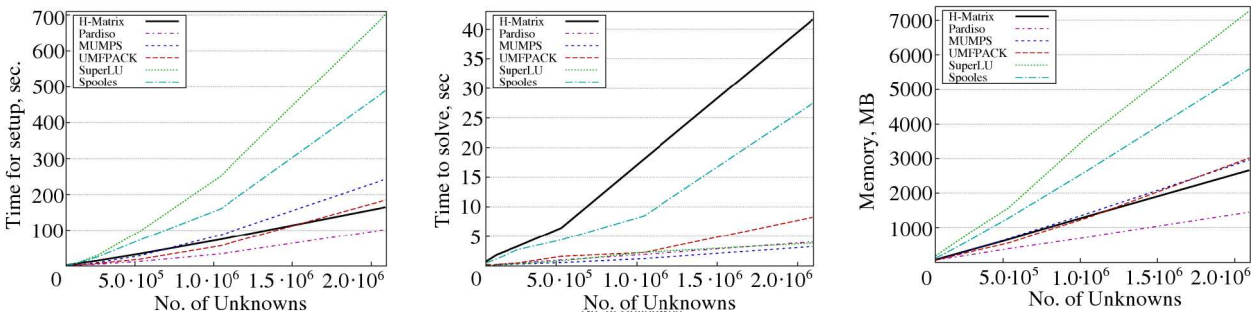

Fig. 5.2. Complexity of setup (left), solve (middle) and memory (right) for the Helmholtz equation in $\mathbb{R}^2$

The solve phase on the other hand again shows the disadvantage of an iterative process compared to direct solving. Athough only up to 4 steps were needed to reach the stopping criterion, the execution time is much larger than for direct solvers.

For the memory consumptions the results are similar compared to the Poisson problem, although the difference between UMFPACK, MUMPS and PARDISO on one side and $\mathcal{H}$-matrices on the other side is smaller. SuperLU and Spooles show the highest memory requirements.

In the case of the convection-diffusion equation (5.3) in $d = 2$, the results for the preprocession and the solve as well as for the memory consumption are almost identical compared to the Poisson equation. The only important difference is the solving time of the algebraic multigrid solver. BoomerAMG needed more iteration steps compared to the Poisson problem and therefore, shows a much higher execution time (see Fig. 5.3).

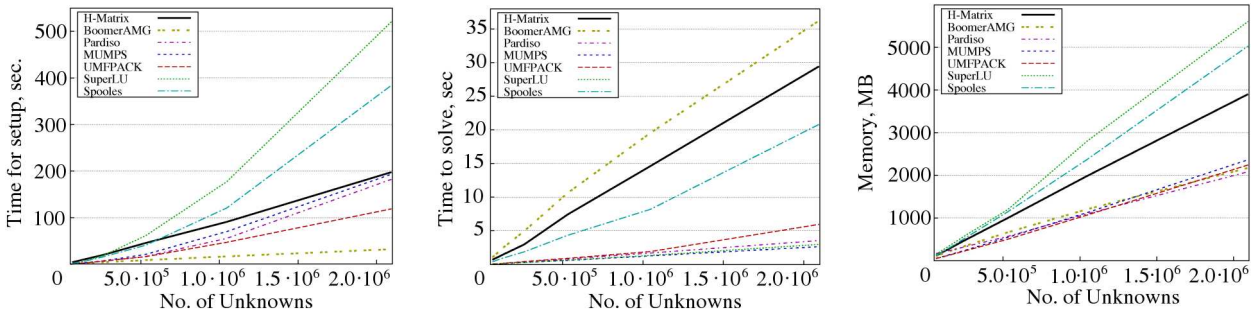

Fig. 5.3. Complexity of setup (left), solve (middle) and memory (right) for convection-diffusion equation in $\mathbb{R}^2$

For the problems in $\mathbb{R}^3$, the scaling behavior of the direct solvers is far less optimal, exceeding quadratic complexity. In contrast to this, the $\mathcal{H}$-matrix solver maintains its almost linear behavior with respect to the problem size. Hence, the point of break even is reached at less than $100,000$ unknowns.

It should also be noted that most direct solvers were not able to compute the larger problems with more than $500,000$ unknowns, either because of the time limit or due to memory limitations.

BoomerAMG showed an excellent performance for the Poisson problem, as can be seen from Fig. 5.4. The time for preprocessing and the memory consumption is unrivaled. Only the time for the solve phase is higher than for the direct solvers due to the iterative nature of the algorithm. The same holds for the $\mathcal{H}$-matrix solver, although the gap is smaller in comparison to the 2d Poisson problem.

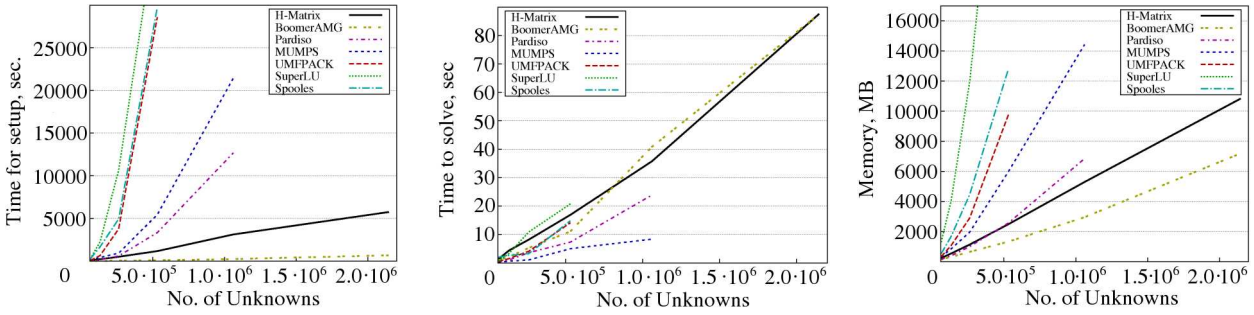

Fig. 5.4. Complexity of setup (left), solve (middle) and memory (right) for the Poisson equation in $\mathbb{R}^3$

For the 3d Helmholtz problem, for which the results are shown in Fig. 5.5, BoomerAMG again did not converge within the preset maximal number of iteration steps. Therefore, the $\mathcal{H}$-matrix solver dominates the direct solvers in terms of the factorization time and memory consumption. For the smaller problems, direct solvers demonstrated a better performance for the solving phase.

In the case of the convection diffusion equation in $\mathbb{R}^3$, the general behavior of the tested solvers remains the same, e.g., almost linear complexity for the algebraic multigrid and $\mathcal{H}$-matrix solver and quadratic complexity for the direct solvers (cf. Fig. 5.6). Most notable is

the solve time and memory consumption of BoomerAMG, which exceeds the corresponding values for $\mathcal{H}$-matrices and also PARDISO. Only the preprocessing is finished slightly earlier by BoomerAMG than by the $\mathcal{H}$-matrix solver.

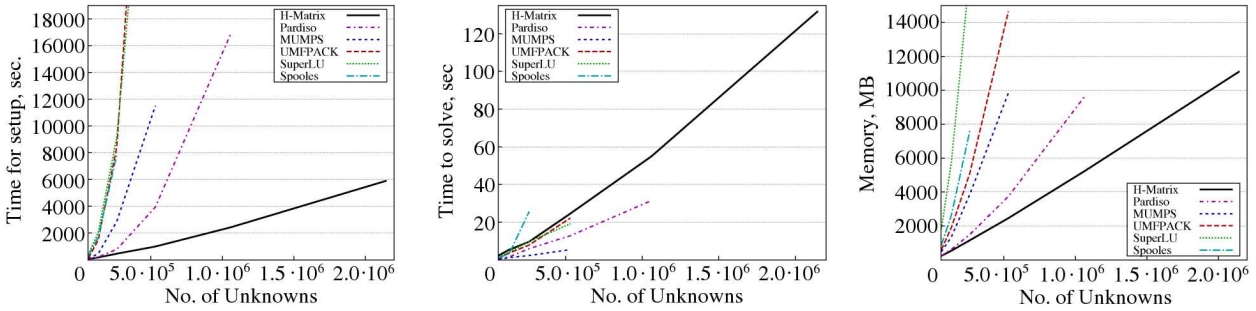

Fig. 5.5. Complexity of setup (left), solve (middle) and memory (right) for the Helmholtz equation in $\mathbb{R}^3$

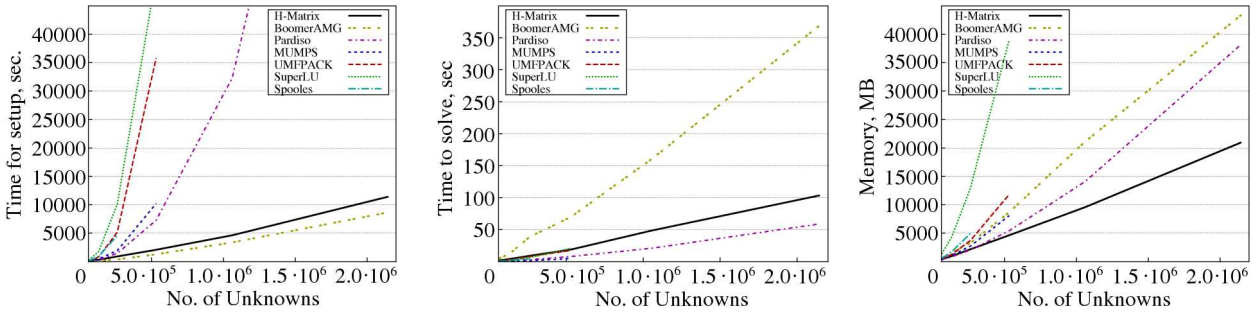

Fig. 5.6. Complexity of setup (left), solve (middle) and memory (right) for convection-diffusion equation in $\mathbb{R}^3$

5.2. $\mathcal{H}$-matrix applicability. Since the number of test matrices is too large to print each result in a table, the concept of a *performance profile* (see [13]) is used to evaluate and compare the performance of each solver. For each solver the performance profile $p(\alpha)$ with respect to a measured result, e.g., time to factorize or solve or the memory consumption, is defined as the fraction of test matrices for which the solver produced a corresponding result within a factor of α to the best result. For $\alpha = 1$, $p(\alpha)$ gives the fraction of matrices yielding the best result using this solver, whereas for $\alpha \to \infty$, $p(\alpha)$ converges to the fraction of matrices, which the solver successfully solved.

All solvers had an upper time limit of three hours for each test matrix. Furthermore, not included in this comparison is the algebraic multigrid method implemented by BoomerAMG, since the small number of successfully solved matrices did not allow a reasonable profile.

Figure 5.7 shows the performance profile for each solver.

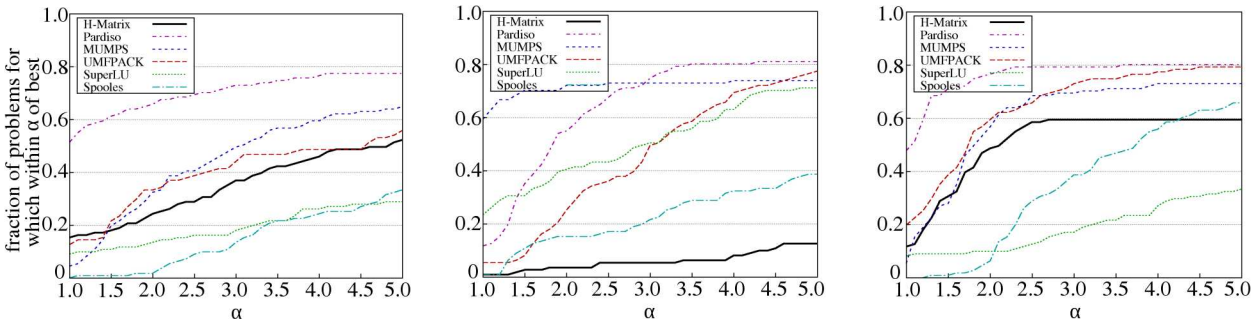

Fig. 5.7. Performance profile for each solver for the preprocessing time (left), the solving time (middle) and the memory consumption (right)

For the preprocessing time, the PARDISO solver demonstrated a superior performance throughout the test set. The $\mathcal{H}$-matrix solver, nevertheless, shows a competitive performance compared to UMFPACK and MUMPS. Although only for a small fraction of all test matrices the best performance is achieved, roughly half of all matrices could be solved. The main problem hindering the $\mathcal{H}$-matrix solver in solving more matrices are singular subblocks detected during factorization. Unfortunately, the $\mathcal{H}$-matrix technique allows no or only minor pivoting. Therefore, if such a singular block is detected, the factorization usually fails. Here further work is needed to develop alternative strategies.

Another interesting aspect is the performance profile for actually solving the system. Here, due to the iterative nature of the solving stage of the $\mathcal{H}$-matrix technique, the performance is not competitive. Especially for a large number of right-hand sides, this plays an important role in the choice of the solver.

In terms of the memory consumption, the $\mathcal{H}$-matrix solver demonstrates a similar performance as most of the direct solvers, especially like UMFPACK and MUMPS, although again, the smaller number of sucessfully solved matrices limits the profile.

To examine the properties of the $\mathcal{H}$-matrix solver for the case of successful applicability, we limit the set of test matrices for the performance profile to those which were solvable with the $\mathcal{H}$-solver. The corresponding profiles are shown in Fig. 5.8.

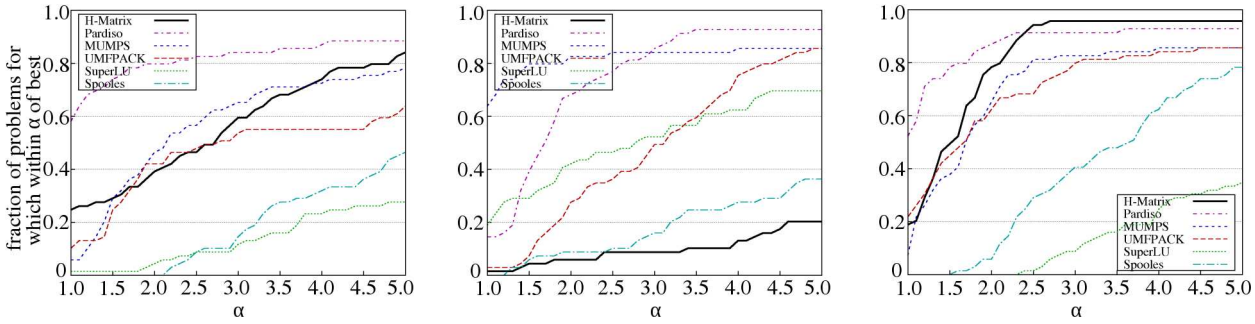

Fig. 5.8. Performance profile for the limited test set for each solver for the preprocessing time (left), the solving time (middle), and the memory consumption (right)

Although again, the PARDISO solver demonstrates the best performance with respect to the preprocessing time in this set of matrices, the results of the $\mathcal{H}$-matrix solver exceeds the corresponding results of the UMFPACK and the MUMPS solver. For the memory consumption, the $\mathcal{H}$-solver even shows the best profile, demonstrating the superior memory complexity of this technique. Only the solving time is inferior due to the iteration technique.

References

1. P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications, **23** (2001), no. 1, pp. 15–41.

2. P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, *Multifrontal parallel distributed symmetricand unsymmetric solvers*, Comput. Methods Appl. Mech. Eng., **184** (2000), pp. 501–520.

3. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel Computing, **32** (2006), no. 2, pp. 136–156.

4. C. Ashcraft and R. Grimes, *SPOOLES: An object-oriented sparse matrix library*, in: *In Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

5. M. Bebendorf, *Why finite element discretizations can be factored by triangular hierarchical matrices*, SIAM J. Num. Anal., **45** (2007), pp. 1472–1494.

6. S. Börm, L. Grasedyck, and W. Hackbusch, *Hierarchical Matrices*, Lecture Note 21 of the Max Planck Institute for Mathematics in the Sciences, Leipzig, Germany, available online at www.mis.mpg.de/preprints/ln/, revised version June 2006, 2003.

7. S. Börm, L. Grasedyck, and W. Hackbusch, *Introduction to hierarchical matrices with applications*, Engineering Analysis with Boundary Elements, **27** (2003), pp. 405–422.

8. S. L. Borne, L. Grasedyck, and R. Kriemann, *Domain-decomposition Based H-LU Preconditioners*, in: *Domain Decomposition Methods in Science and Engineering XVI* (O. Widlund and D. Keyes, eds.), vol. 55 of *Lecture Notes in Computational Science and Engineering*, Springer, 2006, pp. 661–668.

9. A. Brandt, S. McCormick, and J. Ruge, *Algebraic multigrid (AMG) for sparse matrix equations*, in: *Sparsity and its Applications*, Cambridge University Press, 1984, pp. 257–284.

10. O. Bröker, M. Grote, C. Mayer, and A. Reusken, *Robust parallel smoothing for multigrid via sparse approximate inverses*, SIAM J. Sci. Comput., **23** (2001), pp. 1396–1417.

11. T. A. Davis, *Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, **30** (2004), pp. 196–199.

12. J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM J. Matrix Anal. and Appl., **20** (1999), no. 3, pp. 720–755.

13. E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming, **91** (2002), no. 2, pp. 202–213.

14. N. Gibbs, W. Poole, and P. Stockmeyer, *A comparison of several bandwidth and profile reduction algorithms*, ACM Trans of Mathematical Software, **2** (1976), pp. 322–330.

15. L. Grasedyck and W. Hackbusch, *Construction and arithmetics of $\mathcal{H}$-matrices*, Computing, **70** (2003), pp. 295–334.

16. L. Grasedyck, R. Kriemann, and S. L. Borne, *Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems*, Comput. Visual Sci., **11** (2008), pp. 273–291.

17. L. Grasedyck, R. Kriemann, and S. LeBorne, *Parallel black box domain decomposition based $\mathcal{H}$-LU preconditioning*, Tech. Rep. 115, Max Planck Institute for Mathematics in the Sciences, Leipzig, 2005.

18. L. Grasedyck and S. LeBorne, *$\mathcal{H}$-matrix preconditioners in convection-dominated problems*, SIAM J. Mat. Anal., **27** (2006), pp. 1172–1183.

19. G. Haase, M. Kuhn, and S. Reitzinger, *Parallel AMG on distributed memory computers*, SIAM J. Sci. Comp., **24** (2002), no. 2, pp. 410–427.

20. W. Hackbusch, *A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices*, Computing, **62** (1999), pp. 89–108.

21. W. Hackbusch and B. N. Khoromskij, *A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part II: Application to multi-dimensional problems*, Computing, **64** (2000), pp. 21–47.

22. M. Heath, E. Ng, and B. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Reviews, **33** (1991), pp. 420–460.

23. V. E. Henson and U. M. Yang, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, **41** (2002), pp. 155–177.

24. G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, **20** (1999), no. 1, pp. 359–392.

25. S. Le Borne, S. Oliveira, and F. Yang, *$\mathcal{H}$-matrix preconditioners for symmetric saddle-point systems from meshfree discretizations*, Numerical Linear Algebra and Applications, (2006), to appear.

26. M. Lintner, *The eigenvalue problem for the 2d Laplacian in $\mathcal{H}$-matrix arithmetic and application to the heat and wave equation*, Computing, **72** (2004), pp. 293–323.

27. S. Oliveira and F. Yang, *An algebraic approach for $\mathcal{H}$-matrix preconditioners*, Computing, **80** (2007), pp. 169–188.

28. F. Pellegrin, *SCOTCH 5.0 User's guide*, Tech. rep., LaBRI, Université Bordeaux I, 2007.

29. R. W. Ruge and K. Stüben, *Efficient solution of finite difference and finite element equations by algebraic multigrid (AMG)*, in: *Multigrid Methods for Integral and Differential Equations* (H. H. D. J. Paddon, ed.), Clarenden Press Oxford, 1985, pp. 169–212.

30. O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Journal of Future Generation Computer Systems, **20** (2004), pp. 475–487.

31. O. Schenk and K. Gärtner, *On fast factorization pivoting methods for symmetric indefinite systems*, Elec. Trans. Numer. Anal., **23** (2006), pp. 158–179.

32. O. Schenk, K. Gärtner, and W. Fichtner, *Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors*, BIT, **40** (2000), pp. 158–176.